There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection

Ronald Gil*, Hamed Okhravi†, Howard Shrobe*
*MIT CSAIL, Cambridge, MA
Email: {rongil, hes}@csail.mit.edu
†MIT Lincoln Laboratory, Lexington, MA
Email: hamed.okhravi@ll.mit.edu

Abstract—**Memory corruption attacks have been a major vector of compromised computer systems for decades. Memory safety techniques proposed in the literature provide spatial and temporal safety properties to prevent such attacks. Since enforcing full memory safety on legacy languages such as C/C++ often incurs large runtime overhead, recent techniques have created a trade-off between the granularity of enforcement and overhead. By enforcing safety properties at the granularity of allocations instead of objects, these technique incur only a fraction of the overhead. Of particular note, are the recent software-based, so-called *low-fat* schemes, that encode a pointer's bound information in the pointer value itself, thus avoiding a separate metadata store, and additional lookup overhead. In this paper, we show that trading granularity with overhead is insecure. Specifically, we illustrate a new type of attack, which we call Pointer Stretching, that can bypass the recently proposed low-fat schemes using intra-object spatial corruption. Because of the limitations imposed by the low-fat schemes we devise some tricks that allows us to chain ROP gadgets together before a successful stack pivot. We illustrate a real-world exploit against Nginx that successfully hijacks control even when both stack and heap are protected with the software-based low-fat schemes. Furthermore, we show that the vulnerability is inherent in the design of such techniques, and not easily fixable without significant additional implementation and overhead. In addition, we develop an analysis tool to find such vulnerabilities and analyze many popular applications and servers. We find that the exploitable conditions are abundantly found in real-world code bases. Our findings strongly suggest that effective memory defenses must operate at the granularity of objects (and sub-objects) for them to provide meaningful protection against memory corruption attacks.**

I. Introduction

Memory corruption attacks have been used as a major vector of attack against computer systems for decades [1], [2]. Such attacks have evolved from simple code injection attacks [2] to various forms of code reuse (ROP) attacks [3]–[8] as a result of widespread deployment of defenses such as W$\oplus$X in modern operating system [9]. Even though the methods of memory corruption attacks have evolved over the years, the underlying cause has remained the same: the lack of memory safety properties in legacy languages like C/C++.

Memory safety techniques prevent the exploitation of memory bugs by enforcing spatial and temporal safety properties [10]. Some memory safety techniques require manual annotations [11], [12], while others such as SoftBound [13] and CETS [14] automatically annotate the code with the necessary safety checks. Moreover, metadata information (*i.e.*, bounds or temporal IDs) can either be stored in conjunction with each pointer by converting a pointer to a structure in what is known as the *fat pointer* schemes [11], [12], or in a separate memory region as in SoftBound [13]. Since expanding pointers to include metadata information breaks memory compatibility, a so-called *low-fat pointer* technique was proposed by Kwon *et al.* [15] that uses extra/unused bits in a pointer to store bounds information.

Since previous automated techniques (*e.g.*, SoftBound and CETS) impose a large runtime performance overhead on the running code ($\sim$ 2x slowdown on average), recently proposed techniques attempt to trade-off performance overhead with enforcement granularity. Such techniques, for example, enforce spatial bounds on memory allocations instead of individual objects. Of particular note are two recent software-based schemes that protect stack [16] and heap [17] allocations. These techniques leverage the idea of low-fat pointers by encoding the metadata in the pointers themselves; however, unlike the hardware-based scheme proposed by Kwon *et al.* [15], they do not use extra or unused bits for this purpose. Instead, they create a correlation between an allocation's location in memory and its size. For example, in these schemes, allocations in the region of 0x100000000 to 0x200000000 all have a certain size (*e.g.*, 16 bytes), while allocations in the region of 0x200000000 to 0x300000000 all have another size (*e.g.*, 32 bytes). Thus, simply by observing the base address of a pointer, its valid bounds can be calculated without extra information. The implicit hypothesis in these schemes is that allocation protection prevents or "raises the bar" against spatial memory attacks.

In this paper, we refute this hypothesis by demonstrating that allocation protection is too coarse grained to prevent or "raise the bar" for spatial memory corruption attacks. We demonstrate an attack we call *Pointer Stretching* that can generically bypass such schemes. The underlying problem is composite data types in C/C++ (*e.g.*, `structs`). A pointer to a field of a composite

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

data type should have bounds that encompass only that field. This is known as the problem of narrowing [18], [19]. For example, a buffer inside a `struct` should not be allowed to overflow into other elements of that `struct`. However, in the software-based low-fat schemes, due to the correlation between an allocation's location and its bounds, any pointer to a field (or sub-field) of a composite data type gets the same bounds as the entire data type. We illustrate that an attacker can cause spatial corruption inside composite data types to perform control hijacking. Note that while the original hardware-based low-fat scheme can properly narrow each pointer to its appropriate field because the bounds information can be modified independently of the pointer value (we discuss some caveats later in the paper), this flaw is inherent in the design of recent software-based schemes that correlate pointer values with their bounds, and cannot be easily mitigated without some major redesign or alteration of the entire scheme.

While the problem of intra-object corruption and narrowing has been mentioned in the related work [18], we are the first to build full exploits using just intra-object corruption. Doing so uncovers some of the caveats and intricacies of such exploits. Moreover, we are also the first to analyze the prevalence of exploitable conditions, which in itself highlights the importance of this problem.

To illustrate the feasibility of our attack, we present a real-world exploit against Nginx that can successfully hijack application control and launch a malicious shell in the presence of both stack and heap-based protections.

Since the protection schemes initially make it difficult to use the stack as a chaining mechanism for the ROP gadgets, we devise new tricks in our pointer stretching attack. In our exploit, initially each ROP gadgets go directly to the next gadget (via jump or call) without returning to the stack or a trampoline [5]. This continues until enough gadgets are executed to properly modify the stack pointer to an allocated region under our control. We call this strategy *delayed pivoting*. After the stack pointer is modified, the rest of the attack can proceed as a traditional ROP attack.

With this exploit as our motivation, we develop an automated analysis tool to find such vulnerabilities in code bases. In order to assess the prevalence of opportunities for pointer stretching attacks, we use our tool to perform an automated analysis of twenty four commonly used applications and packages. Our results indicate that conditions necessary for pointer stretching attacks, that is function pointers stored in structures that simultaneously contain data buffers, are abundant in many popular applications.

Our contributions are as follows:

- We present an attack, we call *Pointer Stretching*, that generically bypasses allocation protection schemes, and demonstrate against two such defenses that encode meta-data in the pointer values themselves. We thus refute the hypothesis that allocation protection is an effective granularity for preventing spatial memory corruption, and question efficiency gains at the cost of coarser-grained protections.
- We craft a real-world exploit against Nginx that succeed in hijacking control of a machine remotely while protected by *both* the stack-based and the heap-based allocation protection techniques.
- We develop an automated analysis tool to find the conditions necessary for pointer stretching attacks in real-world code bases.
- We use our tool to perform an analysis of the prevalence of such conditions and demonstrate that they exist abundantly in popular applications.

II. Threat Model

The threat model assumed in this paper is that of a remote attacker who tries to exploit a memory bug in the application or server to hijack its control and achieve remote code execution. The application or server under attack is assumed to have a memory bug which permits writing beyond the bounds of a particular buffer (similar to CVE-2016-5017 or CVE-2014-0133). The hardware and operating system are trusted.

We assume the software-based stack [16] and heap [17] protection low-fat schemes are both enabled on the targeted application and all of its linked libraries. Furthermore, we assume that the implementation of these schemes is correct and bug-free. These defenses are really meant to be sufficient on their own, but since protections such as W $\oplus$ X and Address Space Layout Randomization (ASLR) are widely deployed in modern operating systems, we assume that those are enabled as well.

Since the software-based low-fat schemes focus on spatial memory safety, we treat temporal memory violations as out of scope, and do not use them in our attacks.

Our threat model is consistent with related work in the area of memory defense and the threat model assumed by the low-fat schemes.

III. Low-Fat Pointer Techniques

A. Low-Fat Schemes

The original implementation of the low-fat pointer scheme was introduced in hardware [15]. This hardware low-fat scheme provides fine-grained spatial safety while reducing the overhead of fat pointers. It uses 18 bits of a 64-bit word to contain a block size, lowest valid multiple of the block size, and largest valid multiple of the block size. The other 46 bits are used to store the pointer address. These four values are then used to reconstruct the base and bounds. Whenever a computed pointer goes out of bounds, it is permanently changed to be an `Out-of-Bounds Pointer` hardware type. This type of pointer will produce an error if there is ever an attempt to dereference it.

It is important to note that due to the dependence on a fixed block size, this scheme by default offers only an approximation, albeit a relatively accurate one, to the actual bounds. This problem can be remedied by having the compiler pad sub-objects, so that every sub-object is aligned to the exact block size. Note that the bounds can be set independently of the

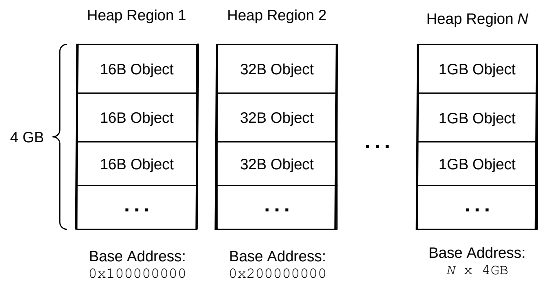


Fig. 1: An overview of how heap regions are designated in the low-fat software scheme to store objects of a particular size.

```c
typedef struct {
        char buf[124];
        void *fptr;
} mystruct;

mystruct s;
```

Fig. 2: A struct vulnerable to intra-object corruption.

```
       ...
       Bounds Checking Instructions
Gadget { mov rsi, qword ptr [rax + 0x80];
       { jmp rsi;
       ...
```

Fig. 3: Selecting a gadget to avoid including bounds checks.

pointer address in the hardware scheme. Thus, a pointer can be properly narrowed to a sub-object's bounds when necessary.

Similar to its hardware counterpart, the recently proposed low-fat pointer software schemes rely on bits stored in the 64-bit pointer representation to determine bounds on stack [16] or heap [17]. However, instead of storing separate bits and reducing the representable address space, they encode the bounds into the address itself. For this setup to work, pointers are assigned to specific address ranges based on the size of the object they point to as shown in Figure 1. Then, pointer arithmetic is instrumented to check the size corresponding to a particular region. Given the size of a region and the fact that all objects in a region are of equal size, the code can then safely determine if pointers resulting from pointer arithmetic are outside of these bounds.

Consequently, this strategy encodes the bounds information quite efficiently and reduces he runtime overhead of safety checks. However, this efficiency comes at a cost to precision compared to the hardware scheme. Because there is a correlation between a pointer's value and its bounds, there is no inherent ability to narrow bounds for sub-objects. Hence, the software scheme is described as only protecting allocation bounds rather than object bounds. We demonstrate that this trade-off between efficiency and security is too weak in most popular code bases.

The idea of protecting allocation bounds instead of object bounds predates the low-fat software pointer scheme. Baggy Bounds [18], PAriCheck [20], Jones and Kelly's GCC patch (J&K) [21], and other schemes directly based on it [22], [23] also protect at the allocation granularity, and are thus vulnerable to our attacks. Note, however, that the attacks are fundamental to the design of the recent software-based low-fat schemes because the correlation between a pointer address and its bounds makes it difficult to fix the inability to establish narrower bounds.

IV. POINTER STRETCHING ATTACK

Since both the stack-based and the heap-based low-fat schemes focus on spatial memory safety, we also illustrate the pointer stretching attack using spatial corruption. A pointer stretching attack[1] starts by causing an intra-object corruption. Consider the code snippet in Figure 2. The `struct` is stored in a region (on stack or heap) that has a designated allocation size. To demonstrate the power of the pointer stretching attack, we can even assume that the allocation size is chosen ideally to fit the `struct`, so that no inter-object corruption is possible. In this case, the `struct` is stored in a region with the designated size of 128 bytes that matches its size perfectly. Any pointer to any field of this `struct` is inherently pointing to the same region where the `struct` is stored, thus it can point to any field of the `struct` even when the protection is enabled. For example, the bounds of `*s.buf` encompass the entire `struct`, while it should legitimately only point to `buf`. As a result, we can overflow the buffer inside `mystruct` to control the function pointer `*ftpr`. Since this modification of the function pointer is done through an overflow, and not the intrinsic instructions in the application itself, it is not instrumented by the low-fat schemes. As a result, after the overflow, `*ftpr` can point to any region in memory. Note again that the underlying weakness is the association between an allocation's address and its bounds which inherently prevents the technique from properly narrowing the bounds of `*s.buf`.

Selecting the gadgets themselves from a hardened binary is not problematic. In most cases, the start of a gadget can be selected to avoid including bounds checking instructions as shown in Figure 3. When the checks cannot be avoided, such as for consecutive memory accesses in one gadget, the only restriction imposed is that the bounds on the start address for any indexing done in the instruction must correspond to the bounds of the resulting address. This scenario is shown in Figure 4. We use both cases in our real-world exploit.

Although it may seem like such an overwrite can be dangerous, what we have at this point is far from a complete attack. The overwritten function pointer can point to a ROP gadget, but for an attack to succeed, we need to be able to run a series of ROP gadgets. Even if the goal is to ultimately run one gadget that issues a system call, for example to launch a shell, we need to be able to set the arguments (%rax, %rdi, %rsi, etc.) properly, which necessitates a chaining mechanism. However, so far we do not have control over the stack (or a

[1]The name refers to the fact that buffer pointer bounds are "stretched" to corrupt other sub-objects stored in a composite data structure.

```
                 ...
                 mov rbx, rdi;
               ⎧ Bounds Checking Instructions
               ⎪ mov rsi, qword ptr [rdi +
       Gadget ⎨ 0x40];
               ⎪ Bounds Checking Instructions
               ⎩ call qword ptr [rdi + 0x30];
                 ...
       bounds(rdi) = bounds(rdi + 0x40) ✓
                   = bounds(rdi + 0x30) ✓
```

Fig. 4: An out of bounds error can be avoided by ensuring accesses in a gadget do not go out of bounds when used.

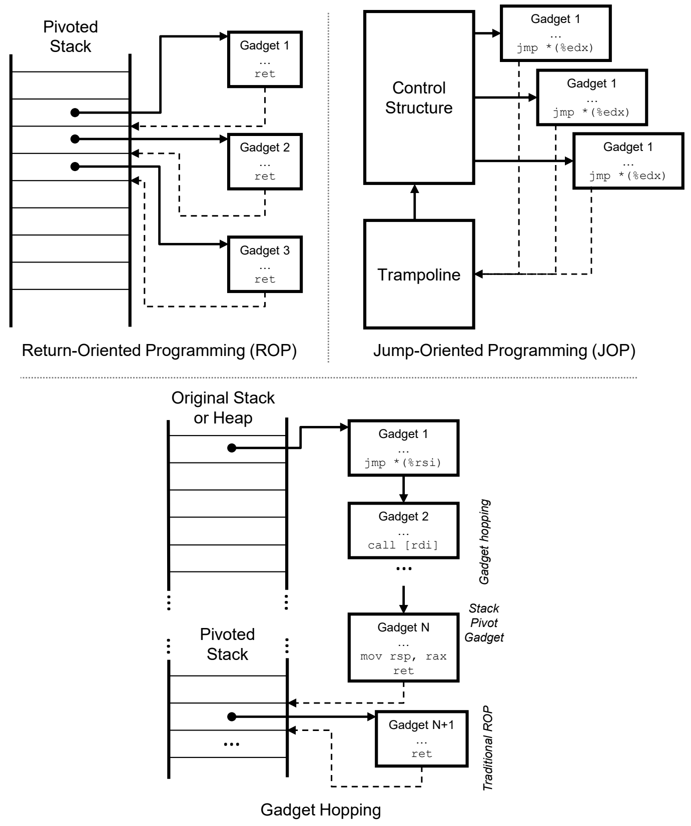


Fig. 5: Gadget hopping compared to traditional ROP and JOP techniques.

trampoline) to chain gadgets together. Moreover, it is unlikely to find a gadget that loads the stack pointer (%rsp) from an area currently under our control (the `struct`). To overcome these challenges, we leverage a trick we call *gadget hopping*.

A. Gadget Hopping

It is possible to find a series of gadgets that ultimately modify %rsp to point to the area under our control (stack pivoting), but we need to initially chain these gadgets somehow without relying on the stack. In previous code reuse attacks, the control always comes back to a central location that contains a list of gadget addresses. This central location is the stack in ROP attacks, and a trampoline in JOP attacks [5], [6]. We observe that this central location is not necessary, particularly for short sequences of gadgets. By carefully selecting gadgets, we can come up with a set of gadgets in a way that each gadget directly transfers control to the next one either through a call or a jump, until the last gadget properly pivots the stack. We call this trick *gadget hopping* that can be used for *delayed pivoting*. Figure 5 illustrates this technique and its comparison with ROP and JOP. In traditional ROP attacks, such a technique is not necessary since the attacker controls the stack and a simple `pop rsp;` gadget can be used for pivoting. However, when the system is protected by allocation protection techniques, gadget hopping is necessary since the area under the control of the attacker is initially very small (*e.g.*, the inside of a `struct`).

Our real-world exploits use three gadgets for gadget hopping. After this step, the attack proceeds as a traditional ROP attack; chaining enough gadgets together to setup the arguments properly and issuing a system call. By issuing a system call with the arguments of their choice, attackers can achieve arbitrarily malicious behavior including launching a shell, creating a backdoor socket, adding an account to a system, *etc.*

V. Real-World Exploits

In this section we present a real-world exploit targeting Nginx using the pointer stretching attack described in the previous section. The attack succeed in the presence of the combined software-based low-fat stack and heap protection schemes [16], [17]. We have also developed a similar attack against Apache, which we do not include in this paper because of space limitations.

The gadgets used in the exploit are all indeed from the hardened binary. Note that after the buffer is overflown in the first step and the code pointer is overwritten, the code pointer can point to any location of our choosing. We can thus skip the any accompanying check added by the protection scheme that may precede each gadget. In other words, for most of the gadgets, the protection provides no impediment after the initial corruption since the bounds checking snippets are just skipped by pointing the code pointer to the instructions following such checks. In a few cases, we use gadgets where that is not possible; for example, we make successive memory accesses in one gadget. In those cases, the checks do not fail since the arithmetic done for the memory accesses is still within the bounds derived from the original address.

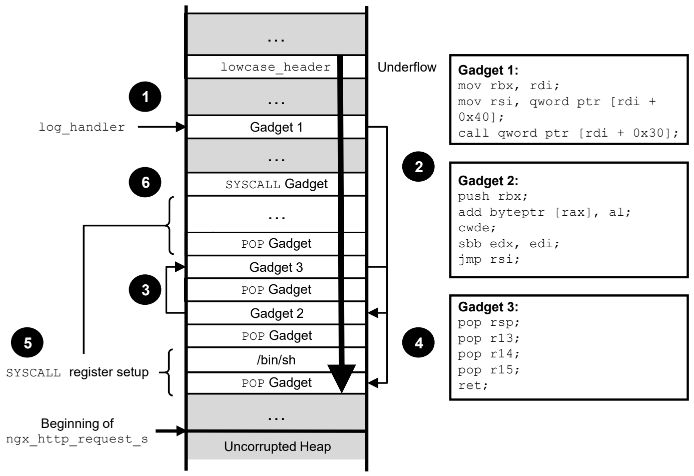
A. Nginx Attack

The objective of this attack is to achieve arbitrary code execution using the pointer stretching attack and demonstrate that allocation protection is an insufficient granularity for bounds checking. We search for a structure containing both a buffer and a function pointer we can corrupt to ultimately execute a shell. After inspecting the source code, we find the structure `ngx_http_request_s`, shown in Figure 6. It is allocated for each request and has both of these features: a buffer, `lowcase_header`, which contains part of the header as it is parsed and a function pointer, `log_handler`, which is called if there is an issue processing the request.

The buffer is located below the function pointer in the structure, so overwriting it requires a buffer underflow. Similar

```c
struct ngx_http_request_s {
    uint32_t                  signature;   /* "HTTP" */
    ngx_connection_t          connection;
    ...
    ngx_http_log_handler_pt  log_handler;
    ...
    u_char    lowcase_header[NGX_HTTP_LC_HEADER_LEN];
    ...
    unsigned                  http_minor:16;
    unsigned                  http_major:16;
};
```

Fig. 6: A snippet of the definition of the `ngx_http_request_s` struct.



Gadget 1:
```
mov rbx, rdi;
mov rsi, qword ptr [rdi +
0x40];
call qword ptr [rdi + 0x30];
```

Gadget 2:
```
push rbx;
add byteptr [rax], al;
cwde;
sbb edx, edi;
jmp rsi;
```

Gadget 3:
```
pop rsp;
pop r13;
pop r14;
pop r15;
ret;
```

Fig. 7: A general overview of the attack and how the stack looks after the stack pivot (i.e. part of the overwritten struct). The `POP` and `SYSCALL` gadget entries refer to a gadgets of the form `INSTRUCTION; ret;`.

memory bugs have been discovered in Nginx in the past, for example `CVE-2009-2629` [24]; we assume such a memory bug exists. This assumption is consistent with related work in this domain [25]–[29] and also the threat model assumed by the software-based low-fat schemes [16], [17].

The underlying weakness arises from the fact that any pointer to any sub-field of the `ngx_http_request_s` can point to the entire `struct`, thus such a buffer underflow is not prevented by the software-based low-fat schemes as long as the corruption is contained in the `struct` and does not corrupt the areas outside of it. Note that the corruption happens purely based on intra-object (`struct`) overwrites, so the function pointer `log_handler` can be corrupted without causing out-of-bound violations.

First, we observe when the corruptible function gets called. Whenever there is any issue processing a request, a worker process calls `ngx_http_log_error`.Towards the end of this function, the corruptible pointer `log_handler` is invoked. To ensure a problem is detected and control is eventually passed to this function, we simply request a nonexistent page.

Now, for the initial step of the attack we overwrite the function pointer `log_handler` with a ROP gadget to do a stack pivot. We need the stack pointer to point to a region of memory we control, so that we can continue executing further instructions. Analyzing the call site reveals that there is a pointer to the same struct we are underflowing (named `r`). At the default optimization level, this pointer ends up stored in a register. Furthermore, it gets passed as the first and second argument to `log_handler` since `ctx->current_request` is an alias to the same pointer. Per x86_64-bit calling convention this setup means registers `rdi` and `rsi` contain pointers to this structure which we control.

Unfortunately, there are no available single stack pivot gadgets for the 64-bit registers we can use here. Nevertheless, we can create a *delayed pivot* by *gadget hopping*. As described earlier, the idea is to create a chain of gadgets that call or jump to addresses set by previous gadgets in the chain. Only the last gadget ends with a `ret` instruction, by which point we have completed the pivot. More concretely, here we can use a chain of three gadgets to achieve the delayed pivot. The first gadget runs three important instructions: move `rdi` (the struct pointer) into `rbx`, move the contents at address `rdi+0x40` into `rsi`, and call the address stored at `rdi+0x30`. Thus, we set `log_handler` to the first gadget, `rdi+0x30` to the second gadget, and `rdi+0x40` to the third gadget as shown in Figure 7. Since `rdi` is simply a pointer to the start of the struct, we use the same original underflow to set all these addresses. Given our setup, after the first gadget executes, the instruction pointer moves to the address stored at `rdi+0x30`, which is the address of the second gadget. The relevant instructions in this gadget are pushing `rbx` onto the stack and then jumping to `rsi`. Since the first gadget set `rsi` to the address at `rdi+0x40`, it now points to the address of the third gadget. This gadget completes the stack pivot by popping `rsp` and then popping three other arbitrary data registers. The sole purpose of popping the extra registers is to move the stack pointer beyond the very beginning of the struct in order to preserve the `connection` member of the struct. This data pointer is used between the location of the underflow and the corrupted call site, so it is easier to simply preserve the value for this exploit.

With the stack pivot complete, we can proceed to execute many more gadgets from the new stack owing to the large size of the struct. In particular, we create a chain of gadgets that sets the arguments for an `execve` system call. In order to do complete the system call though, we need to the base address of the `libc` library to obtain a gadget for the final `syscall` instruction. We obtain this address by using a known libc pointer on the stack. Since we know the offset of this pointer from the base pointer, we add it to the address in the `rbp` register to get the address of the `libc` pointer on the stack. Then, we simply make the address provided for the syscall gadget relative to the known address. Note that `libc` itself is also "hardened" by the low-fat bounds, but those have no impact on our usage of the gadgets, as described earlier.

At this point, anything currently in the data registers is irrelevant going forward, so we reuse some registers used for the stack pivot. Moreover, for simplicity, the remaining system call setup is done through gadgets of the form `instruction; ret;` and we refer to them only by the corresponding instruction. First, we pop off the next value on the (pivoted) stack into register r10. Through the underflow, we set this value to be the string "/bin/sh". Then, we use a gadget to set another register, rdi, to the address of an arbitrary valid object in memory. The only constraint is that it needs to be large enough to hold two pointers. Once we have some valid address in the register, we use a gadget to store the pointer in r10 into the address of rdi. We can use the same gadgets to store a null byte in the address of rdi+8, but since inserting a null byte in the underflow is problematic, we xor a register with itself so that it gets zeroed and execute a mov to the correct address.

Finally, we use further pop gadgets to set the actual arguments to the syscall instruction. This includes setting the execve syscall number of 0x3b in rax, setting rdi to the "/bin/sh" string, rsi to the data pointer that contains the string pointer and NULL, and rdx to that pointer + 8 (so it points to NULL).

We bootstrap the exploit by sending the malicious payload to cause the underflow and consequently overwrite the various parts of the struct. Upon completion, the process image switches to a shell. Through a similar construction, we can execute any arbitrary command restricted only by the permissions given to the process (which depends on the web server configuration).

VI. Analysis Tool

Using the successful exploit as our motivation and in order to obtain an estimate of how prevalent the exploitable conditions for the pointer stretching attack are, we developed a tool to automatically find structures containing both function pointers and buffers. Specifically, we create a Clang plugin to find struct definitions with those characteristics and an LLVM pass to assess the frequency of accesses to such structs. We categorize the accesses to members of these structs as buffer accesses, function pointer accesses, and other accesses. These values do not take into account a member's identity, so in the case of multiple buffers or function pointers, access numbers are aggregated based solely on their type.

A. *Struct Existence*

We use the Clang plugin to find struct definitions that contain both a buffer and a function pointer. The plugin first searches through a package's type declarations for structs. Then, it iterates through the members of those structs to find a buffer and a function pointer. Buffers are found by checking if the type of a member is an array. A pointer is determined to be a function pointer only if it is a pointer directly to a function. The results from this plugin serve as a starting point for finding potential exploitable conditions.

B. *Struct Access*

To calculate the number of times these vulnerable structs are accessed, we use an LLVM pass. The core of the pass' code is shown in Figure 8. It iterates through basic blocks, which represent a single entry single exit piece of code, to find GetElementPointer instructions. These instructions, which occur whenever there is an attempt to get one of a

```
virtual bool runOnBasicBlock(BasicBlock &BB) {
 for (BasicBlock::iterator ii = BB.begin(),
      ii_e = BB.end(); ii != ii_e; ++ii) {
  if (GetElementPtrInst *gep =
      dyn_cast<GetElementPtrInst>(&*ii)) {
   Type *srcElem = gep->getSourceElementType();
   if (StructType *srcStruct =
       dyn_cast<StructType>(srcElem)) {
    // Uses isFunctionPointer
    if (isStructWithBufPtr(srcStruct)) {
     printDebugInfo(&*ii);
     errs() << srcStruct->getName().str() << ": ";
     if (gep->getResultElementType()->isArrayTy()) {
      errs() << "Found struct access to array member."
             << '\n';
     } else if
        (isFunctionPointer(gep->getResultElementType())){
        errs() << "Found struct access to FP member."
               << '\n';
     } else {
        errs() << "Found struct access to other member."
               << '\n';
     }
    }
   }
  }
 }

  return false;
}

bool isFunctionPointer(Type *ptrType) {
 // Recursively follow pointer
 while (PointerType *ptr = dyn_cast<PointerType>(ptrType)){
  Type* pointeeType = ptr->getTypeAtIndex((unsigned)0);
  if (pointeeType->isFunctionTy()) return true;
  ptrType = pointeeType;
 }

  return false;
}
```

Fig. 8: Core snippet of the LLVM pass used to find accesses into vulnerable structures.

struct's members, are then inspected in a similar manner to the Clang plugin. We iterate through the struct being accessed to check whether it contains both a function pointer and an array. The array is detected by simply checking that the member's type is an array. In the LLVM pass, the check for a function pointer is done recursively. In other words, the pointer is recursively followed to check whether the final reference is to a function. If the access is to a relevant struct, then the type of the member being accessed is examined in order to correctly categorize the access.

C. *Unsound Analysis*

The technique we use to detect function pointers is unsound. Alternatively, we could have used an analysis that provided soundness guarantees such as LLVM's alias analysis. However, these algorithms tend to be conservative in order to provide soundness guarantees. Thus, they are likely to provide false positives for ambiguous cases. In this scenario, we prefer to have an underestimation instead of an overestimation of the exploitable conditions. Thereby, the results provided here represent a lower bound on the prevalence of these structs and accesses in the analyzed code bases.

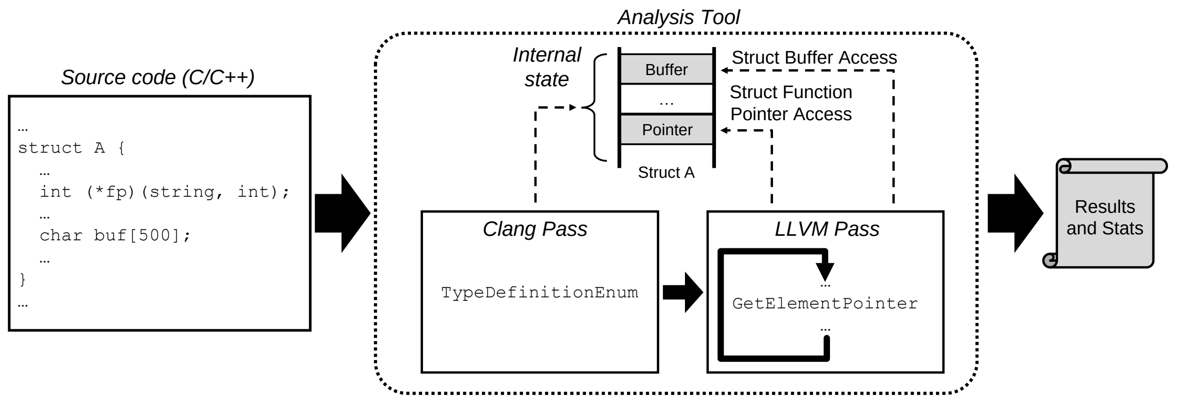


Fig. 9: Vulnerable `struct` finding tool: a Clang plugin (definitions) and an LLVM pass (usage).

TABLE I: A summary of `structs` containing a buffer and a function pointer. The top section is a selection of widely used programs, web servers, and databases. The bottom section is the set of top installed C source packages according to Ubuntu's Popularity Contest [30].

	Version	Definition	Buffer Access	Fptr Access	Other Access	Total Access
git	2.12.0	8	140	51	965	1156
httpd	2.4.25	7	68	15	842	925
lighttpd	1.4	1	0	7	468	475
nginx	1.10.3	3	55	60	2789	2904
openssl	1.4.45	6	6	217	2674	2897
postgresql	9.6	8	1	26	709	736
redis	3.2.7	1	0	5	10	15
acl	2.2.52	0	0	0	0	0
bash	4.4	2	3	14	76	93
coreutils	8.26	1	6	2	19	27
e2fsprogs	1.43.3	6	13	235	5446	5694
findutils	4.6.0	0	0	0	0	0
gcc	6.3	1	0	2	62	64
grep	2.27	3	10	6	90	106
gzip	1.6	0	0	0	0	0
hostname	3.18	0	0	0	0	0
ncurses	6.0	2	68	37	1158	1263
pam	1.1.3	1	0	1	33	34
perl	5.20.2	0	0	0	0	0
sed	4.4	1	0	0	0	0
slang	2.3.0	8	103	103	953	1159
tar	1.29	0	0	0	0	0
util-linux	2.19	1	0	3	24	27
zlib	1.2.8	0	0	0	0	0

D. Tool Usage

In order to use our analysis tool, we compile packages using Clang and provide the proper flags in order to invoke the plugin and the pass. Both the plugin and the pass are set to output their findings to `stderr`, so we redirect `stderr` to a file in order to collect the output and parse it. The tool is also set to print extra debug information if there is any available (using the `-g` flag in Clang). This extra output is useful when later checking the data against the source for further inspection or analysis. The overall flow of the analysis is shown in Figure 9.

The results of these analyses are shown in Table I. Note that for some packages, there appear to be no buffer accesses despite the existence of a vulnerable `struct`. This scenario is often caused by the buffer being modified exclusively by some external function such as `memset` or `memcpy` which goes undetected by the pass. As can be observed, `structs` containing buffers and function pointers are prevalently used in popular applications and servers. This finding strongly suggests that effective memory safety must be at least at the granularity of sub-objects, and that allocation bounds checking does not provide sufficient protection against control hijacking attacks.

VII. Possible Countermeasures

Our attacks could be prevented by incorporating full memory safety defenses that ensure spatial and temporal safety properties on pointers at the granularity of sub-objects. SoftBound [13] and CETS [14] are two such techniques providing spatial and temporal pointer safety properties, respectively.

Another option for composite data structures is to automatically allocate sub-objects separately and keep only pointers to these sub-objects in the original composite object. However, this approach has various issues. One is that it requires an extra memory access for any sub-object member of a composite object. It can also incur significant memory overhead since the object containing the pointers to sub-objects still needs to be of full size to maintain alignment. Furthermore, it may impose a significant performance penalty due to the poor locality of sub-objects and potentially reduced cache hit ratio.

A potential improvement to this strategy is to store only vulnerable arrays separately rather than all sub-objects. This idea then raises the question of how to accurately determine what arrays are vulnerable. Thus, let us assume any array in a vulnerable structure has the potential to be vulnerable through some programmer error. The precise memory overhead and performance impact would depend on the specific runtime usage of an application, but at least some index can be surmised from the number of problematic buffers accessed in an application. There will be one more memory dereference for every buffer access, so the number of buffer accesses shown in Table I would essentially double in terms of memory accesses.

As a variation of storing only vulnerable arrays separately, the arrays can instead be stored contiguously but with added padding such that the bounds lie on protected object bounds. This approach is only practical when the array's parent structure is small or primarily composed of the array. The reason is that the array takes as much space as the parent structure since the corresponding bounds to the memory section will be the same ones used for the parent structure. Nevertheless, this approach has the potential to increase locality and thereby cache effectiveness and overall performance at the expense of greater memory overhead. In this case the memory overhead can potentially be as high as $2 + n$ times the original usage where n is the number of vulnerable arrays in the structure. Thus, in the best case the overhead will be 3x the size of the parent structure.

Alternatively, just the metadata for each sub-object could be stored in a separate region referenced by indexes stored in those data structures. While this technique prevents our attack, it also adds possibly significant overhead to the bounds checking schemes by essentially making them more similar to SoftBound (with a separate metadata region) than the fat-pointer schemes. Such a remediation would also add memory overhead to the scheme. A hybrid approach that uses the pure low-fat scheme for non-compound objects and a metadata based scheme for compound objects is also a possibility.

In general, the software-based low-fat schemes' strict dependence on pointer addresses to determine bounds makes it difficult to modify them so as to provide sub-object protection while remaining a pure low-fat approach. Either sub-objects need to be moved to different addresses or extra information independent of the pointer address needs to be stored somewhere in memory. These options increase time and space overheads and/or diverge away from the fat pointer strategy.

VIII. Conclusion

In this paper, we have illustrated a new type of attack that can bypass bounds checking spatial memory safety techniques that protect allocations. We used intra-object corruption to overwrite code pointers and hijack control. We illustrated our attack using a real-world exploit. We further developed an automated tool and analyzed the vulnerable conditions in many popular applications and packages, and found that such conditions are prevalent in the wild. Our findings indicate that allocation protection does not provide sufficient defense against spatial memory corruption attacks, and the trade-off between the granularity of protection and the performance gain contributes to major security weakness in such schemes.

References

[1] J. P. Anderson, "Computer security technology planning study. volume 2," DTIC Document, Tech. Rep., 1972.

[2] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[3] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of CCS*, 2007.

[4] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. of IEEE SP*, 2013.

[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. of CCS*, 2010.

[6] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proc. of CCS*, 2011.

[7] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *36th*, ser. S&P, 2015.

[8] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, C. L. Stephen Crane, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity," in *Proc. of NDSS*, 2017.

[9] OpenBSD. (2003) Openbsd 3.3. OpenBSD. [Online]. Available: http://www.openbsd.org/33.html

[10] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. of IEEE Symposium on Security and Privacy*, 2013.

[11] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," *ACM SIGPLAN Notices*, vol. 37, no. 1, 2002.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[14] ——, "CETS: compiler enforced temporal safety for C," ser. ISMM, 2010.

[15] A. Kwon, U. Dhawan, J. Smith, T. Knight, and A. Dehon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proc. of CCS*, 2013.

[16] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *Proc. of NDSS*, 2017.

[17] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *Proc. of International Conference on Compiler Construction*, 2016.

[18] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[19] intel. (2013) Introduction to intel memory protection extensions. Intel. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions?language=es

[20] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: An efficient pointer arithmetic checker for c programs," in *Proc. of ASIACCS*, 2010.

[21] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Proc. of the 3rd International Workshop on Automatic Debugging*, 1997.

[22] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector." in *Proc. of NDSS*, 2004.

[23] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[24] "Cve-2009-2629," 2009. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2009-2629

[25] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming," in *Proc. of NDSS*, 2015.

[26] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," 2014.

[27] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. of IEEE SP*, 2015.

[28] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. of USENIX Sec*, 2015.

[29] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proc. of ACM CCS*, 2015.

[30] A. Pennarun, B. Allombert, and P. Reinhold. (2012, April) Ubuntu popularity contest. [Online]. Available: http://popcon.ubuntu.com/